Lattice QCD GPU programming with Quda and Grid

October 31, 2025

renwick.james.hudspith@gmail.com

Outline

- *BLAS interface in QUDA
  - using the interface
  - parameters
  - ignoring the interface
- Intro to GRID
  - compiling
  - command line arguments
  - shortcomings and optimizations

What we learned last lecture

Writing an interface to QUDA is very easy, and feels like the usual steps in CUDA.

Implementing your own kernel in QUDA is a little painful because of the templating neccessary.

Batched BLAS routines are more performant on the GPU than individual BLAS calls, this is valuable for our contractions (particularly DFTs) as they can be written as N operations of small-matrix $\times$ bigger matrix

An example from slaph_interface.cpp

```
int nInBlock = 0 , blockStart = 0 ;
for (int dil1=0; dil1<n1; dil1++) {
  cpu_quark_bar_param.v = host_quark_bar[dil1] ;
  ColorSpinorField quark_bar(cpu_quark_bar_param) ;
  ColorSpinorField quda_quark_bar(cuda_quark_bar_param) ;
  quda_quark_bar = quark_bar ;
  for (int dil2=0; dil2<n2; dil2++){
    innerProductQuda( quda_quark_bar, quda_quark[dil2],
        (std::complex<double>*)d_tmp+n_sites*nInBlock );
    nInBlock++ ;
    if( nInBlock == blockSizeMomProj ) {
    blas_lapack::native::stridedBatchGEMM( d_mom, d_tmp,
     (std::complex<double>*)d_ret+blockStart*X[3]*nMom,
            cublas_param_mom_sum, QUDA_CUDA_FIELD_LOCATION);
    blockStart += nInBlock ; nInBlock = 0 ;
}}}
```

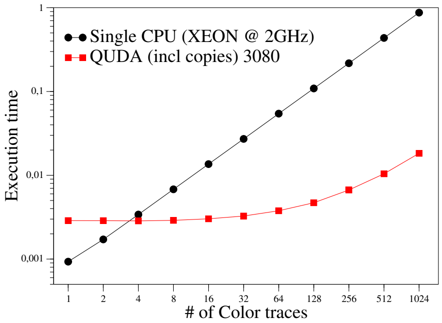
This zoom-in only concerns what is happening on the device, other copies to and from the host are assumed.

We allocate "quda_quark " normally with n2 indices, but for dil1 "quda_quark_bar " carries only "dil1" index so we load and delete in the dil1 loop saving memory and improving data locality. These contracted products go into "d_tmp " a device-temporary piece of memory.

The inner products are performed for every dil2 and dil1 index and the DFT is done for all T and all momenta at once. Going into "d_ret", which will be sent back to the host.

We will focus on the strided batched GEMM QUDA interface for the next part of this lecture...

Results from the homework colorContract 16^4 local volume on Ernest



Single CPU computation scales linearly with number of traces. GPU computation is at first limited only by transfer speeds and finally at large enough N starts being more compute bound.
Single CPU is very weak and more realistic architectures would push the black line to the right. GPU in double precision is also weak.

We will look at my fork of QUDA: (https://github.com/RJHudspith/quda) because of this issue https://github.com/lattice/quda/issues/1554. At the moment QUDA's batched BLAS implementations do not translate 1-to-1 to those in cuBLAS, whereas mine do. QUDA supports currently only dense LU inversion and strided, batched GEMM calls with cuBLAS cuBLAS and hipBLAS. Other simpler BLAS options are handled natively - such as axpy's.

There are two places we should now consider - the interface "`/lib/interface/blas_interface.cpp`" and the device implementations "`/lib/targets/*/blas_lapack.cpp`"

By default QUDA's ccmake will link cuBLAS or hipBLAS so we don't really have to do any more steps at compilation time.

Let's look at the interface

```
void blasGEMMQuda( const void *arrayA,
                   const void *arrayB,
                   void *arrayC,
                   const QudaBoolean use_native,
                   QudaBLASParam blas_param)
```

This does the batched, strided operation
$C = \alpha op(arrayA) op(arrayB) + \beta arrayC$ where A, B, and C are flattened arrays of the matrices we are multiplying. `use_native` tells us whether the arrays are on the device or on the host and need to be copied.
`blas_param` are the QUDA parameters

Contents of BLASparam struct

```
typedef struct QudaBLASParam_s {
    size_t struct_size;
    QudaBLASType blas_type; /**< Type of BLAS computation to perfrom */
    QudaBLASOperation trans_a; /**< operation op(A) that is non- or (conj.) transpose. */
    QudaBLASOperation trans_b; /**< operation op(B) that is non- or (conj.) transpose. */
    int m;                     /**< number of rows of matrix op(A) and C. */
    int n;                     /**< number of columns of matrix op(B) and C. */
    int k;                     /**< number of columns of op(A) and rows of op(B). */
    int lda;                   /**< leading dimension of A. */
    int ldb;                   /**< leading dimension of B. */
    int ldc;                   /**< leading dimension of C */
    int a_stride;              /**< stride of the A array in strided(batched) mode */
    int b_stride;              /**< stride of the B array in strided(batched) mode */
    int c_stride;              /**< stride of the C array in strided(batched) mode */
    double_complex alpha; /**< scalar used for multiplication. */
    double_complex beta;  /**< scalar used for multiplication. If beta==0, C does not have
    int inv_mat_size; /**< The rank of the square matrix in the LU inversion */
    int batch_count;              /**number of batched multiplies to perform*/
    QudaBLASDataType data_type;   /**< Specifies if using S(C) or D(Z) BLAS type */
    QudaBLASDataOrder data_order; /**< Specifies if using Row or Column major */
  } QudaBLASParam;
```

Some things that are nice here are that we can specify our data order and QUDA will perform transposes to translate to cuBLAS's Fortran order. This is beneficial because we typically have everything in row-major c-order.

The data type can be float/double real or complex.

The interface for non-native (host) applications just does the usual allocating and freeing of memory as well as memcpys to and from the device.

Ultimately this is a very shallow, convenient interface to underlying cuBLAS routines if we already are linking to QUDA.

Let's skip the interface

Instead if we have stuff already on the device we can call directly

```
blas_lapack::native::stridedBatchGEMM(arrayA,
                                      arrayB,
                                      arrayC,
                                      blas_param,
                                      QUDA_CUDA_FIELD_LOCATION);
```

If we pass anything other than the CUDA field location the code will error exit

And that's about it. As this casts through the void (pretty much the default for QUDA interfaces) we have to be careful about our BLAS parameters (n,m,k,lda,ldb,ldc,a_stride,b_stride,c_stride) not trying to access memory that is unavailable.

Grid is an OpenMP-like interface

Besides the differences in configuration and running, Grid's interface with the GPU is opaque.

Grid takes the view that a group of CPUs is an "accelerator" just like a GPU is and each MPI rank sees an "accelerator". Most of the underlying GPU-stuff is hidden from the user but it is very more-or-less what we've seen before. For instance their reduction is like the one presented in Lecture 1.

This is much like how QDP++ works, where each operation e.g. a product of propagators, is it's own multithreaded and enclosed job. This is hideously inefficient for simple operations as we really want multiple operations to happen in the same parallel region. This becomes much worse when we reach GPU offloading.

Intro to Grid

Grid takes its name from splitting the local subvolume into a smaller grid that puts a lexicographical index into a separate SIMD lane so that things like 3x3 matrix operations have perfect SIMD parallelism. Such that data ordering for a matrix is U[c1][c2][x] with x running fastest. This is opposite to QDP++ which is (color minor) U[x][c1][c2] and cannot easily be extended to even AVX.

This mapping extends a little to the GPU where `threadIdx.x` is the number of SIMD lanes (usually 2, 4, or 8) and the y-threads are user-supplied. These will index the local volume.

The nice thing about grid is that the same exact code can be developed and used for the CPU and the GPU! As we will see in a bit this also translates to optimisations too!

Compiling and running GRID on Perlmutter

For the CPU configure command I use:

```
./configure CXX=mpic++ CXXFLAGS=-std=c++17
--enable-comms=mpi --enable-simd=AVXFMA
```

And for the GPU

```
./configure CXX=nvcc MPICXX=CC CXXFLAGS='-ccbin CC
-gencode arch=compute_80,code=sm_80
-std=c++17 -cudart shared'
LDFLAGS='-cudart shared'
--enable-comms=mpi --enable-simd=GPU
--enable-gen-simd-width=64
--enable-shm=nvlink
--enable-accelerator=cuda
--enable-unified=no
```

renwick.james.hudspith@gmail.com

If in doubt consult:
https://github.com/paboyle/Grid/tree/develop/systems as likely someone has done some profiling for you

Grid usually requires some command line flags for running:

```
--accelerator-threads 32 --comms-overlap --shm 2048 --shm-mpi 0
```

chief of which is the "accelerator threads" argument, there is no autotuning here but typically a number between 8 and 64 is good. Overlapping comms and compute is very important for a GPU due to the penalties with off-node comms. The other arguments don't matter as much.

Grid leaves it up to the user to make devices visible through a binding script.

A cautionary tale

As GRID is dividing the local volume we will typically want to use even-odd preconditioning we have to be very careful not to end up with an odd number of sites in any one direction. We will hit an assert on startup if we do.

For example, I wanted to run a $28^3 \times 56$ HMC run on 2 PerlMutter nodes so local volume $(10) \times 14 \times 14 \times 28 \times 28$ and this works for the double precision particioning (1.1.2.2 SIMD) but for single (1.2.2.2 partitioning) we end up with an odd "reduced" dimension in y. In the end I extended the time direction to 64.

Let's look at the Stout smearing code

```
void smear(GaugeField& u_smr, const GaugeField& U) const {
  GaugeField C(U.Grid());
  GaugeLinkField tmp(U.Grid()), iq_mu(U.Grid()), Umu(U.Grid())
  u_smr = U ; // set the smeared field to the current gauge fi
  SmearBase->smear(C, U);
  for (int mu = 0; mu < Nd; mu++) {
    if( mu == OrthogDim ) continue ;
    Umu = peekLorentz(U, mu);
    tmp = peekLorentz(C, mu);
    iq_mu = Ta( tmp * adj(Umu));
    exponentiate_iQ(tmp, iq_mu);
    pokeLorentz(u_smr, tmp * Umu, mu);
  }
};
```

"C" has the staples. The `exponentiate_iQ` is a problem here.

```
void exponentiate_iQ(GaugeLinkField& e_iQ,
                const GaugeLinkField& iQ) const {
  GridBase* grid = iQ.Grid();
  GaugeLinkField unity(grid);
  unity = 1.0;
  GaugeLinkField iQ2(grid),iQ3(grid);
  LatticeComplex u(grid),w(grid),f0(grid),f1(grid),f2(grid);
  iQ2 = iQ * iQ;
  iQ3 = iQ * iQ2;
  set_uw(u, w, iQ2, iQ3);
  set_fj(f0, f1, f2, u, w);
  e_iQ = f0 * unity + timesMinusI(f1) * iQ - f2 * iQ2;
};
```

Many lattice temporaries, parallel region opened and shut all the time. Everything is complex when many temporaries are purely real. *It is concise though.*

What is happening under the hood

Each operation has an "accelerator for" loop hidden inside which is defined with macros:

```
#define accelerator_forNB( iter1, num1, nsimd, ... ) \
  accelerator_for2dNB( iter1, num1, iter2, 1, nsimd,\
   {__VA_ARGS__} );
#define accelerator_for( iter, num, nsimd, ... )  \
  accelerator_forNB(iter, num, nsimd, { __VA_ARGS__ } ); \
  accelerator_barrier(dummy)
```

Which is calling the 2D interface of CUDA where it interprets `__VA__ARGS__` as a lambda function, which is quite cute! Note that this is only working with blockIdx.x, blockIdx.y is 1

What's going on with iter2? Turns out nothing

```
#define accelerator_for2dNB( iter1, num1, iter2, num2, nsimd, ..
  {         \
    if ( num1*num2 ) {       \
      int nt=acceleratorThreads();     \
      typedef uint64_t Iterator;     \
      auto lambda = [=] accelerator     \
 (Iterator iter1,Iterator iter2,Iterator lane) mutable {  \
        __VA_ARGS__;     \
      };       \
      dim3 cu_threads(nsimd,acceleratorThreads(),1);   \
      dim3 cu_blocks ((num1+nt-1)/nt,num2,1);    \
      LambdaApply<<<cu_blocks,cu_threads,0,computeStream>>>\
       (num1,num2,nsimd,lambda); \
    }         \
  }
```

Back to matrix exponentiation

```
void exponentiate_iQ( GaugeLinkField &e_iQ,
                      const GaugeLinkField &iQ) const {
e_iQ = 1.0 ;
{
   autoView( e_iQ_v , e_iQ  , AcceleratorWrite ) ;
   autoView( iQ_v   , iQ    , AcceleratorRead ) ;
   accelerator_for(ss,e_iQ_v.size(), \
                   GaugeLinkField::vector_object::Nsimd(),\
   { const auto iQ2 = iQ_v[ss]*iQ_v[ss] ;
     auto u = -imag(trace(iQ2*iQ_v[ss]))*0.3333333333333333148;
     auto w = -real(trace(iQ2))*0.5;
     auto f0 = 0.3849001794597505244*w ;
     w = sqrt(w) ;
     f0 = f0*w ;
     f0 = acos(u/f0)*0.3333333333333333148;
```

renwick.james.hudspith@gmail.com

```
    u = w*(0.57735026918962573l1)*cos(f0);
    w = w*sin(f0);
    auto f2 = timesI( sin(w)/w );
    auto u2 = u * u; auto w2 = w * w; w = cos(w);
    const auto emiu = cos(u) - timesI(sin(u));
    u = 2.*u ;
    auto e2iu = cos(u) + timesI(sin(u));
    f0 = e2iu*(u2-w2)+emiu*((8.0*u2*w)+(u*(3.0*u2+w2)*f2));
    auto f1 = e2iu*u-emiu*((u*w)-(3.0*u2-w2)*f2);
    f2 = e2iu - emiu * (w + (1.5*u) * f2);
    w = 1.0 ;
    w = w / (9.0 * u2 - w2);  // reals
    f0 = f0 * w ;
    f1 = f1 * w ;
    f2 = f2 * w ;
    e_iQ_v[ss]=f0*e_iQ_v[ss]+timesMinusI(f1)*iQ_v[ss]-f2*iQ2;
  });}
```

Dissecting my optimisation

Note that we create an "autoView" (creating new vector-length device variables e_iQ_v and iQ_v) where we are sending iQ to the device, and writing `e_iQ` from the device to the host. The other temporaries are local to the device and for that index. To improve cache locality we reuse parameters as much as possible and "auto" the variables that will alias to real or complex appropriately. We define an iterator index "ss" which will be our local site index.

This code is more than $2\times$ faster than GRID's implementation for CPU threads this translates directly to the GPU because the principle is the same (but more dramatic). e.g. single-GPU (16^4 volume) on Perlmutter in ms:

eiQ_old 2.125 vs eiQ_new 0.187

Applying the same techniques to the smeared forces and to the staples gave us an overall $10\times$ speedup for our smeared forces for our DWF ensemble generation. This stout smearing on the CPU is still slower than GLU's though....

What we have learned

To utilise the GPU we benefit most with doing mathematically intensive operations and keeping stuff on the device and doing local operations on the device. We should try and hide the overhead with compute.

The overhead of opening a parallel region (openMP CPU) or launching a kernel (GPU) (+ copies) is high, minimising this by doing more work in the parallel region will always provide a benefit. This is the same with QDP++ on the CPU and why its mutlithreading was always shit

The batched-BLAS interface is in QUDA and available to be used, this is the key to optimal contractions. As `quda_laph` links against QUDA we might as well use it.

GRID is like QDP++ where we can write high-level code for the CPU that will just work on the GPU. It has some mechanisms for optimising hot code which are often neccessary to use.